PIPELINE ACCELERATOR FOR IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD

CLAIM OF PRIORITY

[1] This application claims priority to U.S. Provisional Application Serial No. 60/422,503, filed on October 31, 2002, which is incorporated by reference.

CROSS REFERENCE TO RELATED APPLICATIONS

[2] This application is related to U.S. Patent App. Serial Nos. ___ entitled IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD (Attorney Docket No. 1934-11-3), _____ entitled COMPUTING MACHINE HAVING IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD (Attorney Docket No. 1934-12-3), _____ entitled PROGRAMMABLE CIRCUIT AND RELATED COMPUTING MACHINE AND METHOD (Attorney Docket No. 1934-14-3), and _______ entitled PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE UNITS AND RELATED COMPUTING MACHINE AND METHOD (Attorney Docket No. 1934-15-3), which have a common filing date and owner and which are incorporated by reference.

BACKGROUND

[3] A common computing architecture for processing relatively large amounts of data in a relatively short period of time includes multiple interconnected processors that share the processing burden. By sharing the processing burden, these multiple processors can often process the data more quickly than a single processor can for a given clock frequency. For example, each of the processors can process a respective portion of the data or execute a respective portion of a processing algorithm.

[4] **FIG. 1** is a schematic block diagram of a conventional computing machine ***10*** having a multi-processor architecture. The machine ***10*** includes a master processor ***12*** and coprocessors $\boldsymbol{14_1} - \boldsymbol{14_n}$, which communicate with each other and the master processor via a bus ***16***, an input port ***18*** for receiving raw data from a remote device (not shown in **FIG. 1**), and an output port ***20*** for providing processed data to the remote source. The machine ***10*** also includes a memory ***22*** for the master processor ***12***, respective memories ***24***$_1$ – ***24***$_n$ for the coprocessors ***14***$_1$ – ***14***$_n$, and a memory ***26*** that the master processor and coprocessors share via the bus ***16***. The memory ***22*** serves as both a program and a working memory for the master processor ***12***, and each memory ***24***$_1$ – ***24***$_n$ serves as both a program and a working memory for a respective coprocessor ***14***$_1$ – ***14***$_n$. The shared memory ***26*** allows the master processor ***12*** and the coprocessors ***14*** to transfer data among themselves, and from/to the remote device via the ports ***18*** and ***20***, respectively. The master processor ***12*** and the coprocessors ***14*** also receive a common clock signal that controls the speed at which the machine ***10*** processes the raw data.

[5] In general, the computing machine ***10*** effectively divides the processing of raw data among the master processor ***12*** and the coprocessors ***14***. The remote source (not shown in **FIG. 1**) such as a sonar array loads the raw data via the port ***18*** into a section of the shared memory ***26***, which acts as a first-in-first-out (FIFO) buffer (not shown) for the raw data. The master processor ***12*** retrieves the raw data from the memory ***26*** via the bus ***16***, and then the master processor and the coprocessors ***14*** process the raw data, transferring data among themselves as necessary via the bus ***16***. The master processor ***12*** loads the processed data into another FIFO buffer (not shown) defined in the shared memory ***26***, and the remote source retrieves the processed data from this FIFO via the port ***20***.

[6] In an example of operation, the computing machine ***10*** processes the raw data by sequentially performing n + 1 respective operations on the raw data, where these operations together compose a processing algorithm such as a Fast Fourier Transform (FFT). More specifically, the machine ***10*** forms a data-processing pipeline from the master processor ***12*** and the coprocessors ***14***. For a given frequency of the clock signal, such a pipeline often allows the machine ***10*** to process the raw data faster than a machine having only a single processor.

[7] After retrieving the raw data from the raw-data FIFO (not shown) in the memory ***26***, the master processor ***12*** performs a first operation, such as a trigonometric function, on the raw data. This operation yields a first result, which the processor ***12*** stores in a first-result FIFO (not shown) defined within the memory ***26***. Typically, the processor ***12*** executes a program stored in the memory ***22***, and performs the above-described actions under the control of the program. The processor ***12*** may also use the memory ***22*** as working memory to temporarily store data that the processor generates at intermediate intervals of the first operation.

[8] Next, after retrieving the first result from the first-result FIFO (not shown) in the memory ***26***, the coprocessor $\boldsymbol{14_1}$ performs a second operation, such as a logarithmic function, on the first result. This second operation yields a second result, which the coprocessor $\boldsymbol{14_1}$ stores in a second-result FIFO (not shown) defined within the memory ***26***. Typically, the coprocessor $\boldsymbol{14_1}$ executes a program stored in the memory $\boldsymbol{24_1}$, and performs the above-described actions under the control of the program. The coprocessor $\boldsymbol{14_1}$ may also use the memory $\boldsymbol{24_1}$ as working memory to temporarily store data that the coprocessor generates at intermediate intervals of the second operation.

[9] Then, the coprocessors $\boldsymbol{24_2} - \boldsymbol{24_n}$ sequentially perform third – nth operations on the second – (n-1)th results in a manner similar to that discussed above for the coprocessor $\boldsymbol{24_1}$.

[10] The nth operation, which is performed by the coprocessor $\boldsymbol{24_n}$, yields the final result, *i.e.*, the processed data. The coprocessor $\boldsymbol{24_n}$ loads the processed data into a processed-data FIFO (not shown) defined within the memory ***26***, and the remote device (not shown in **FIG. 1**) retrieves the processed data from this FIFO.

[11] Because the master processor ***12*** and coprocessors ***14*** are simultaneously performing different operations of the processing algorithm, the computing machine ***10*** is often able to process the raw data faster than a computing machine having a single processor that sequentially performs the different operations. Specifically, the single processor cannot retrieve a new set of the raw data until it performs all n + 1 operations on the previous set of raw data. But using the pipeline technique discussed above, the master processor ***12*** can retrieve a new set of raw data after performing only the first operation. Consequently, for a given clock frequency, this pipeline technique can increase the speed at which the machine ***10*** processes the raw data by a factor of approximately n + 1 as compared to a single-processor machine (not shown in **FIG. 1**).

[12] Alternatively, the computing machine ***10*** may process the raw data in parallel by simultaneously performing n + 1 instances of a processing algorithm, such as an FFT, on the raw data. That is, if the algorithm includes n + 1 sequential operations as described above in the previous example, then each of the master processor ***12*** and the coprocessors ***14*** sequentially perform all n + 1 operations on respective sets of the raw data. Consequently, for a given clock frequency, this parallel-processing technique, like the above-described pipeline technique, can increase the speed at which the machine ***10*** processes the raw data by a factor of approximately n + 1 as compared to a single-processor machine (not shown in **FIG. 1**).

[13] Unfortunately, although the computing machine ***10*** can process data more quickly than a single-processor computer machine (not shown in **FIG. 1**), the data-processing speed of the machine ***10*** is often significantly less than the frequency of the processor clock. Specifically, the data-processing speed of the computing machine ***10*** is limited by the time that the master processor ***12*** and coprocessors ***14*** require to process data. For brevity, an example of this speed limitation is discussed in conjunction with the master processor ***12***, although it is understood that this discussion also applies to the coprocessors ***14***. As discussed above, the master processor ***12*** executes a program that controls the processor to manipulate data in a desired manner. This program includes a sequence of instructions that the processor ***12*** executes. Unfortunately, the processor ***12*** typically requires multiple clock cycles to execute a single instruction, and often must execute multiple instructions to process a single value of data. For example, suppose that the processor ***12*** is to multiply a first data value A (not shown) by a second data value B (not shown). During a first clock cycle, the processor ***12*** retrieves a multiply instruction from the memory ***22***. During second and third clock cycles, the processor ***12*** respectively retrieves A and B from the memory ***26***. During a fourth clock cycle, the processor ***12*** multiplies A and B, and, during a fifth clock cycle, stores the resulting product in the memory ***22*** or ***26*** or provides the resulting product to the remote device (not shown). This is a best-case scenario, because in many cases the processor ***12*** requires additional clock cycles for overhead tasks such as initializing and closing counters. Therefore, at best the processor ***12*** requires five clock cycles, or an average of 2.5 clock cycles per data value, to process A and B..

[14] Consequently, the speed at which the computing machine ***10*** processes data is often significantly lower than the frequency of the clock that drives the master processor ***12*** and the coprocessors ***14***. For example, if the processor ***12*** is clocked at 1.0 Gigahertz (GHz) but requires an average of 2.5 clock cycles per data value, then the effective data-processing speed equals (1.0 GHz)/2.5 = 0.4 GHz. This effective data-processing speed is often characterized in units of operations per second. Therefore, in this example, for a clock speed of 1.0 GHz, the processor ***12*** would be rated with a data-processing speed of 0.4 Gigaoperations/second (Gops).

[15] **FIG. 2** is a block diagram of a hardwired data pipeline ***30*** that can typically process data faster than a processor can for a given clock frequency, and often at substantially the same rate at which the pipeline is clocked. The pipeline ***30*** includes operator circuits $\boldsymbol{32_1} - \boldsymbol{32_n}$,which each perform a respective operation on respective data without executing program instructions. That is, the desired operation is "burned in" to a circuit ***32*** such that it implements the operation automatically, without the need of program instructions. By eliminating the overhead associated with executing program instructions, the pipeline ***30*** can typically perform more operations per second than a processor can for a given clock frequency.

[16] For example, the pipeline ***30*** can often solve the following equation faster than a processor can for a given clock frequency:

$$Y(x_k) = (5x_k + 3)2^{xk}$$

where x_k represents a sequence of raw data values. In this example, the operator circuit $\boldsymbol{32_1}$ is a multiplier that calculates $5x_k$, the circuit $\boldsymbol{32_2}$ is an adder that calculates $5x_k + 3$, and the circuit $\boldsymbol{32_n}$ (n = 3) is a multiplier that calculates $(5x_k + 3)2^{xk}$.

[17] During a first clock cycle k=1, the circuit ***32_1*** receives data value x_1 and multiplies it by 5 to generate $5x_1$.

[18] During a second clock cycle k = 2, the circuit ***32_2*** receives $5x_1$ from the circuit ***32_1*** and adds 3 to generate $5x_1 + 3$. Also, during the second clock cycle, the circuit ***32_1*** generates $5x_2$.

[19] During a third clock cycle k = 3, the circuit ***32_3*** receives $5x_1 + 3$ from the circuit ***32_2*** and multiplies by 2^{x1} (effectively left shifts $5x_1 + 3$ by x_1) to generate the first result $(5x_1 + 3)2^{x1}$. Also during the third clock cycle, the circuit ***32_1*** generates $5x_3$ and the circuit ***32_2*** generates $5x_2 + 3$.

[20] The pipeline ***30*** continues processing subsequent raw data values x_k in this manner until all the raw data values are processed.

[21] Consequently, a delay of two clock cycles after receiving a raw data value x_1 — this delay is often called the latency of the pipeline ***30*** — the pipeline generates the result $(5x_1 + 3)2^{x1}$, and thereafter generates one result — *e.g.*, $(5x_2 + 3)2^{x2}$, $(5x_3 + 3)2^{x3}$, . . ., $5x_n + 3)2^{xn}$ — each clock cycle.

[22] Disregarding the latency, the pipeline ***30*** thus has a data-processing speed equal to the clock speed. In comparison, assuming that the master processor ***12*** and coprocessors ***14*** (**FIG. 1**) have data-processing speeds that are 0.4 times the clock speed as in the above example, the pipeline ***30*** can process data 2.5 times faster than the computing machine ***10*** (**FIG. 1**) for a given clock speed.

[23] Still referring to **FIG. 2**, a designer may choose to implement the pipeline ***30*** in a programmable logic IC (PLIC), such as a field-programmable gate array (FPGA), because a PLIC allows more design and modification flexibility than does an application specific IC (ASIC). To configure the hardwired connections within a PLIC, the designer merely sets interconnection-configuration registers disposed within the PLIC to predetermined binary states. The combination of all these binary states is often called "firmware." Typically, the designer loads this firmware into a nonvolatile memory (not shown in **FIG. 2**) that is coupled to the PLIC. When one "turns on" the PLIC, it downloads the firmware from the memory into the interconnection-configuration registers. Therefore, to modify the functioning of the PLIC, the designer merely modifies the firmware and allows the PLIC to download the modified firmware into the interconnection-configuration registers. This ability to modify the PLIC by merely modifying the firmware is particularly useful during the prototyping stage and for upgrading the pipeline ***30*** "in the field".

[24] Unfortunately, the hardwired pipeline ***30*** may not be the best choice to execute algorithms that entail significant decision making, particularly nested decision making. A processor can typically execute a nested-decision-making instruction (*e.g.,* a nested conditional instruction such as "if A, then do B, else if C, do D, . . ., else do n") approximately as fast as it can execute an operational instruction (*e.g.,* "A + B") of comparable length. But although the pipeline ***30*** may be able to make a relatively simple decision (*e.g.,* "A > B?") efficiently, it typically cannot execute a nested decision (*e.g.*, "if A, then do B, else if C, do D, . . ., else do n") as efficiently as a processor can. One reason for this inefficiency is that the pipeline ***30*** may have little on-board memory, and thus may need to access external working/instruction memory (not shown). And although one may be able to design the pipeline ***30*** to execute such a nested decision, the size and complexity of the required circuitry often makes such a design impractical, particularly where an algorithm includes multiple different nested decisions.

[25] Consequently, processors are typically used in applications that require significant decision making, and hardwired pipelines are typically limited to "number crunching" applications that entail little or no decision making.

[26] Furthermore, as discussed below, it is typically much easier for one to design/modify a processor-based computing machine, such as the computing machine ***10*** of **FIG. 1**, than it is to design/modify a hardwired pipeline such as the pipeline ***30*** of **FIG. 2**, particularly where the pipeline ***30*** includes multiple PLICs.

[27] Computing components, such as processors and their peripherals (*e.g.,* memory), typically include industry-standard communication interfaces that facilitate the interconnection of the components to form a processor-based computing machine.

[28] Typically, a standard communication interface includes two layers: a physical layer and a services layer.

[29] The physical layer includes the circuitry and the corresponding circuit interconnections that form the interface and the operating parameters of this circuitry. For example, the physical layer includes the pins that connect the component to a bus, the buffers that latch data received from the pins, and the drivers that drive signals onto the pins. The operating parameters include the acceptable voltage range of the data signals that the pins receive, the signal timing for writing and reading data, and the supported modes of operation (*e.g.*, burst mode, page mode). Conventional physical layers include transistor-transistor logic (TTL) and RAMBUS.

[30] The services layer includes the protocol by which a computing component transfers data. The protocol defines the format of the data and the manner in which the component sends and receives the formatted data. Conventional communication protocols include file-transfer protocol (FTP) and transmission control protocol/internet protocol (TCP/IP).

[31] Consequently, because manufacturers and others typically design computing components having industry-standard communication interfaces, one can typically design the interface of such a component and interconnect it to other computing components with relatively little effort. This allows one to devote most of his time to designing the other portions of the computing machine, and to easily modify the machine by adding or removing components.

[32] Designing a computing component that supports an industry-standard communication interface allows one to save design time by using an existing physical-layer design from a design library. This also insures that he/she can easily interface the component to off-the-shelf computing components.

[33] And designing a computing machine using computing components that support a common industry-standard communication interface allows the designer to interconnect the components with little time and effort. Because the components support a common interface, the designer can interconnect them via a system bus with little design effort. And because the supported interface is an industry standard, one can easily modify the machine. For example, one can add different components and peripherals to the machine as the system design evolves, or can easily add/design next-generation components as the technology evolves. Furthermore, because the components support a common industry-standard service layer, one can incorporate into the computing machine's software an existing software module that implements the corresponding protocol. Therefore, one can interface the components with little effort because the interface design is essentially already in place, and thus can focus on designing the portions (*e.g.*, software) of the machine that cause the machine to perform the desired function(s).

[34] But unfortunately, there are no known industry-standard services layers for components, such as PLICs, used to form hardwired pipelines such as the pipeline ***30*** of **FIG. 2**.

[35] Consequently, to design a pipeline having multiple PLICs, one typically spends a significant amount of time and exerts a significant effort designing and debugging the services layer of the communication interface between the PLICs "from scratch." Typically, such an ad hoc services layer depends on the parameters of the data being transferred between the PLICs. Likewise, to design a pipeline that interfaces to a processor, one would have to spend a significant amount of time and exert a significant effort in designing and debugging the services layer of the communication interface between the pipeline and the processor from scratch.

[36] Similarly, to modify such a pipeline by adding a PLIC to it, one typically spends a significant amount of time and exerts a significant effort designing and debugging the services layer of the communication interface between the added PLIC and the existing PLICs. Likewise, to modify a pipeline by adding a processor, or to modify a computing machine by adding a pipeline, one would have to spend a significant amount of time and exert a significant effort in designing and debugging the services layer of the communication interface between the pipeline and processor.

[37] Consequently, referring to **FIGS. 1** and **2**, because of the difficulties in interfacing multiple PLICs and in interfacing a processor to a pipeline, one is often forced to make significant tradeoffs when designing a computing machine. For example, with a processor-based computing machine, one is forced to trade number-crunching speed and design/modification flexibility for complex decision-making ability. Conversely, with a hardwired pipeline-based computing machine, one is forced to trade complex-decision-making ability and design/modification flexibility for number-crunching speed. Furthermore, because of the difficulties in interfacing multiple PLICs, it is often impractical for one to design a pipeline-based machine having more than a few PLICs. As a result, a practical pipeline-based machine often has limited functionality. And because of the difficulties in interfacing a processor to a PLIC, it would be impractical to interface a processor to more than one PLIC. As a result, the benefits obtained by combining a processor and a pipeline would be minimal.

[38] Therefore, a need has arisen for a new computing architecture that allows one to combine the decision-making ability of a processor-based machine with the number-crunching speed of a hardwired-pipeline-based machine.

SUMMARY

[39] According to an embodiment of the invention, a pipeline accelerator includes a memory and a hardwired-pipeline circuit coupled to the memory. The hardwired-pipeline circuit is operable to receive data, load the data into the memory, retrieve the data from the memory, process the retrieved data, and provide the processed data to an external source.

[40] According to another embodiment of the invention, the hardwired-pipeline circuit is operable to receive data, process the received data, load the processed data into the memory, retrieve the processed data from the memory, and provide the retrieved processed data to an external source.

[41] Where the pipeline accelerator is coupled to a processor as part of a peer-vector machine, the memory facilitates the transfer of data — whether unidirectional or bidirectional — between the hardwired-pipeline circuit and an application that the processor executes.

BRIEF DESCRIPTION OF THE DRAWINGS

[42] **FIG. 1** is a block diagram of a computing machine having a conventional multi-processor architecture.

[43] **FIG. 2** is a block diagram of a conventional hardwired pipeline.

[44] **FIG. 3** is a block diagram of a computing machine having a peer-vector architecture according to an embodiment of the invention.

[45] **FIG. 4** is a block diagram of the pipeline accelerator of **FIG. 3** according to an embodiment of the invention.

[46] **FIG. 5** is a block diagram of the hardwired-pipeline circuit and the data memory of **FIG. 4** according to an embodiment of the invention.

[47] **FIG. 6** is a block diagram of the memory-write interfaces of the communication shell of **FIG. 5** according to an embodiment of the invention.

[48] **FIG. 7** is a block diagram of the memory-read interfaces of the communication shell of **FIG. 5** according to an embodiment of the invention.

[49] **FIG. 8** is a block diagram of the pipeline accelerator of **FIG. 3** according to another embodiment of the invention.

[50] **FIG. 9** is a block diagram of the hardwired-pipeline circuit and the data memory of **FIG. 8** according to an embodiment of the invention.

DETAILED DESCRIPTION

[51] **FIG. 3** is a schematic block diagram of a computing machine ***40***, which has a peer-vector architecture according to an embodiment of the invention. In addition to a host processor ***42***, the peer-vector machine ***40*** includes a pipeline accelerator ***44***, which performs at least a portion of the data processing, and which thus effectively replaces the bank of coprocessors ***14*** in the computing machine ***10*** of **FIG. 1**. Therefore, the host-processor ***42*** and the accelerator ***44*** (or units thereof as discussed

below) are "peers" that can transfer data vectors back and forth. Because the accelerator ***44*** does not execute program instructions, it typically performs mathematically intensive operations on data significantly faster than a bank of coprocessors can for a given clock frequency. Consequently, by combining the decision-making ability of the processor ***42*** and the number-crunching ability of the accelerator ***44***, the machine ***40*** has the same abilities as, but can often process data faster than, a conventional computing machine such as the machine ***10***. Furthermore, as discussed below, providing the accelerator ***44*** with a communication interface that is compatible with the communication interface of the host processor ***42*** facilitates the design and modification of the machine ***40***, particularly where the processor's communication interface is an industry standard. And where the accelerator ***44*** includes multiple pipeline units (*e.g.*, PLIC-based circuits), providing each of these units with the same communication interface facilitates the design and modification of the accelerator, particularly where the communication interfaces are compatible with an industry-standard interface. Moreover, the machine ***40*** may also provide other advantages as described below and in the previously cited patent applications.

[52] Still referring to **FIG. 3**, in addition to the host processor ***42*** and the pipeline accelerator ***44***, the peer-vector computing machine ***40*** includes a processor memory ***46***, an interface memory ***48***, a bus ***50***, a firmware memory ***52***, an optional raw-data input port ***54***, a processed-data output port ***58***, and an optional router ***61***.

[53] The host processor ***42*** includes a processing unit ***62*** and a message handler ***64***, and the processor memory ***46*** includes a processing-unit memory ***66*** and a handler memory ***68***, which respectively serve as both program and working memories for the processor unit and the message handler. The processor memory ***46*** also includes an accelerator-configuration registry ***70*** and a message-configuration registry ***72***, which store respective configuration data that allow the host processor ***42*** to configure the functioning of the accelerator ***44*** and the format of the messages that the message handler ***64*** sends and receives.

[54] The pipeline accelerator ***44*** is disposed on at least one PLIC (not shown) and includes hardwired pipelines ***74_1*** – ***74_n***, which process respective data without executing program instructions. The firmware memory ***52*** stores the configuration firmware for the accelerator ***44***. If the accelerator ***44*** is disposed on multiple PLICs, these PLICs and their respective firmware memories may be disposed in multiple pipeline units (**FIG. 4**). The accelerator ***44*** and pipeline units are discussed further below and in previously cited U.S. Patent App. Serial No. ___ entitled PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE UNITS AND RELATED COMPUTING MACHINE AND METHOD (Attorney Docket No. 1934-15-3). Alternatively, the accelerator ***44*** may be disposed on at least one ASIC, and thus may have internal interconnections that are unconfigurable. In this alternative, the machine ***40*** may omit the firmware memory ***52***. Furthermore, although the accelerator ***44*** is shown including multiple pipelines ***74***, it may include only a single pipeline. In addition, although not shown, the accelerator ***44*** may include one or more processors such as a digital-signal processor (DSP). Moreover, although not shown, the accelerator ***44*** may include a data input port and/or a data output port.

[55] The general operation of the peer-vector machine ***40*** is discussed in previously cited U.S. Patent App. Serial No. ___ entitled IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD (Attorney Docket No. 1934-11-3), and the structure and operation of the pipeline accelerator ***44*** is discussed below in conjunction with **FIGS. 4 – 9**.

[56] **FIG. 4** is a schematic block diagram of the pipeline accelerator ***44*** of **FIG. 3** according to an embodiment of the invention.

[57] The accelerator ***44*** includes one or more pipeline units ***78***, each of which includes a pipeline circuit ***80***, such as a PLIC or an ASIC. As discussed further below and in previously cited U.S. Patent App. Serial No. ___ entitled PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE UNITS AND RELATED COMPUTING MACHINE AND METHOD (Attorney Docket No. 1934-15-3), each pipeline unit ***78*** is a "peer" of the host processor ***42*** and of the other pipeline units of the accelerator ***44***.

That is, each pipeline unit ***78*** can communicate directly with the host processor ***42*** or with any other pipeline unit. Thus, this peer-vector architecture prevents data "bottlenecks" that otherwise might occur if all of the pipeline units ***78*** communicated through a central location such as a master pipeline unit (not shown) or the host processor ***42***. Furthermore, it allows one to add or remove peers from the peer-vector machine ***40*** (**FIG. 3**) without significant modifications to the machine.

[58] The pipeline circuit ***80*** includes a communication interface ***82***, which transfers data between a peer, such as the host processor ***42*** (**FIG. 3**), and the following other components of the pipeline circuit: the hardwired pipelines $\boldsymbol{74_1}$-$\boldsymbol{74_n}$ (**FIG. 3**) via a communication shell ***84***, a controller ***86***, an exception manager ***88***, and a configuration manager ***90***. The pipeline circuit ***80*** may also include an industry-standard bus interface ***91***. Alternatively, the functionality of the interface ***91*** may be included within the communication interface ***82***.

[59] By designing the components of the pipeline circuit ***80*** as separate modules, one can often simplify the design of the pipeline circuit. That is, one can design and test each of these components separately, and then integrate them much like one does when designing software or a processor-based computing system (such as the system ***10*** of **FIG. 1**). In addition, one can save in a library (not shown) hardware description language (HDL) that defines these components — particularly components, such as the communication interface ***82***, that will probably be used frequently in other pipeline designs — thus reducing the design and test time of future pipeline designs that use the same components. That is, by using the HDL from the library, the designer need not redesign previously implemented components "from scratch", and thus can focus his efforts on the design of components that were not previously implemented, or on the modification of previously implemented components. Moreover, one can save in the library HDL that defines multiple versions of the pipeline circuit ***80*** or of the entire pipeline accelerator ***44***, so that one can pick and choose among existing designs.

[60] The communication interface ***82*** sends and receives data in a format recognized by the message handler ***64*** (**FIG. 3**), and thus typically facilitates the design and modification of the peer-vector machine ***40*** (**FIG. 3**). For example, if the data format is an industry standard such as the Rapid I/O format, then one need not design a custom interface between the host processor ***42*** and the accelerator ***44***. Furthermore, by allowing the pipeline circuit ***80*** to communicate with other peers, such as the host processor ***42*** (**FIG. 3**), via the pipeline bus ***50*** instead of via a non-bus interface, one can change the number of pipeline units ***78*** by merely connecting or disconnecting them (or the circuit cards that hold them) to the pipeline bus instead of redesigning a non-bus interface from scratch each time a pipeline unit is added or removed.

[61] The hardwired pipelines ***74_1-74_n*** perform respective operations on data as discussed above in conjunction with **FIG. 3** and in previously cited U.S. Patent App. Serial No. __ entitled IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD (Attorney Docket No. 1934-11-3), and the communication shell ***84*** interfaces the pipelines to the other components of the pipeline circuit ***80*** and to circuits (such as a data memory ***92*** discussed below) external to the pipeline circuit.

[62] The controller ***86*** synchronizes the hardwired pipelines ***74_1.74_n*** and monitors and controls the sequence in which they perform the respective data operations in response to communications, *i.e.*, "events," from other peers. For example, a peer such as the host processor ***42*** may send an event to the pipeline unit ***78*** via the pipeline bus ***50*** to indicate that the peer has finished sending a block of data to the pipeline unit and to cause the hardwired pipelines ***74_1.74_n*** to begin processing this data. An event that includes data is typically called a message, and an event that does not include data is typically called a "door bell." Furthermore, as discussed below in conjunction with **FIG. 5**, the pipeline unit ***78*** may also synchronize the pipelines ***74_1.74_n*** in response to a synchronization signal.

[63] The exception manager ***88*** monitors the status of the hardwired pipelines ***74_1-74_n***, the communication interface ***82***, the communication shell ***84***, the controller ***86***, and the bus interface ***91***, and reports exceptions to the host processor ***42*** (**FIG. 3**). For example, if a buffer in the communication interface ***82*** overflows, then the exception manager ***88*** reports this to the host processor ***42***. The exception manager may also correct, or attempt to correct, the problem giving rise to the exception. For example, for an overflowing buffer, the exception manager ***88*** may increase the size of the buffer, either directly or via the configuration manager ***90*** as discussed below.

[64] The configuration manager ***90*** sets the soft configuration of the hardwired pipelines ***74_1-74_n***, the communication interface ***82***, the communication shell ***84***, the controller ***86***, the exception manager ***88***, and the interface ***91***in response to soft-configuration data from the host processor ***42*** (**FIG. 3**) — as discussed in previously cited U.S. Patent App. Serial No. _ entitled IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD (Attorney Docket No. 1934-11-3), the hard configuration denotes the actual topology, on the transistor and circuit-block level, of the pipeline circuit ***80***, and the soft configuration denotes the physical parameters (*e.g.*, data width, table size) of the hard-configured components. That is, soft configuration data is similar to the data that can be loaded into a register of a processor (not shown in **FIG. 4**) to set the operating mode (*e.g.,* burst-memory mode) of the processor. For example, the host processor ***42*** may send soft-configuration data that causes the configuration manager ***90*** to set the number and respective priority levels of queues in the communication interface ***82***. The exception manager ***88*** may also send soft-configuration data that causes the configuration manager ***90*** to, *e.g.*, increase the size of an overflowing buffer in the communication interface ***82***.

[65] Still referring to **FIG. 4**, in addition to the pipeline circuit ***80***, the pipeline unit ***78*** of the accelerator ***44*** includes the data memory ***92***, an optional communication bus ***94***, and, if the pipeline circuit is a PLIC, the firmware memory ***52*** (**FIG. 3**).

[66] The data memory ***92*** buffers data as it flows between another peer, such as the host processor ***42*** (**FIG. 3**), and the hardwired pipelines ***74_1-74_n***; and is also a working memory for the hardwired pipelines. The communication interface ***82*** interfaces the data memory ***92*** to the pipeline bus ***50*** (via the communication bus ***94*** and industry-standard interface ***91*** if present), and the communication shell ***84*** interfaces the data memory to the hardwired pipelines ***74_1-74_n***.

[67] The industry-standard interface ***91*** is a conventional bus-interface circuit that reduces the size and complexity of the communication interface ***82*** by effectively offloading some of the interface circuitry from the communication interface. Therefore, if one wishes to change the parameters of the pipeline bus ***50*** or router ***61*** (**FIG. 3**), then he need only modify the interface ***91*** and not the communication interface ***82***. Alternatively, one may dispose the interface ***91*** in an IC (not shown) that is external to the pipeline circuit ***80***. Offloading the interface ***91*** from the pipeline circuit ***80*** frees up resources on the pipeline circuit for use in, *e.g.*, the hardwired pipelines $\boldsymbol{74_1}$-$\boldsymbol{74_n}$ and the controller ***86***. Or, as discussed above, the bus interface ***91*** may be part of the communication interface ***82***.

[68] As discussed above in conjunction with **FIG. 3**, where the pipeline circuit ***80*** is a PLIC, the firmware memory ***52*** stores the firmware that sets the hard configuration of the pipeline circuit. The memory ***52*** loads the firmware into the pipeline circuit ***80*** during the configuration of the accelerator ***44***, and may receive modified firmware from the host processor ***42*** (**FIG. 3**) via the communication interface ***82*** during or after the configuration of the accelerator. The loading and receiving of firmware is further discussed in previously cited U.S. Patent App. Serial No. ___ entitled PROGRAMMABLE CIRCUIT AND RELATED COMPUTING MACHINE AND METHOD (Attorney Docket No. 1934-14-3).

[69] Still referring to **FIG. 4**, the pipeline circuit ***80***, data memory ***92***, and firmware memory ***52*** may be disposed on a circuit board or card ***98***, which may be plugged into a pipeline-bus connector (not shown) much like a daughter card can be plugged into a slot of a mother board in a personal computer (not shown). Although not shown, conventional ICs and components such as a power regulator and a power sequencer may also be disposed on the card ***98*** as is known.

[70] Further details of the structure and operation of the pipeline unit ***78*** are discussed below in conjunction with **FIG. 5**.

[71] **FIG. 5** is a block diagram of the pipeline unit ***78*** of **FIG. 4** according to an embodiment of the invention. For clarity, the firmware memory ***52*** is omitted from **FIG.**

5. The pipeline circuit ***80*** receives a master CLOCK signal, which drives the below-described components of the pipeline circuit either directly or indirectly. The pipeline circuit ***80*** may generate one or more slave clock signals (not shown) from the master CLOCK signal in a conventional manner. The pipeline circuit ***80*** may also a receive a synchronization signal SYNC as discussed below.

[72] The data memory ***92*** includes an input dual-port-static-random-access memory (DPSRAM) ***100***, an output DPSRAM ***102***, and an optional working DPSRAM ***104***.

[73] The input DPSRAM ***100*** includes an input port ***106*** for receiving data from a peer, such as the host processor ***42*** (**FIG. 3**), via the communication interface ***82***, and includes an output port ***108*** for providing this data to the hardwired pipelines ***74_1-74_n*** via the communication shell ***84***. Having two ports, one for data input and one for data output, increases the speed and efficiency of data transfer to/from the DPSRAM ***100*** because the communication interface ***82*** can write data to the DPSRAM while the pipelines ***74_1-74_n*** read data from the DPSRAM. Furthermore, as discussed above, using the DPSRAM ***100*** to buffer data from a peer such as the host processor ***42*** allows the peer and the pipelines ***74_1-74_n*** to operate asynchronously relative to one and other. That is, the peer can send data to the pipelines ***74_1-74_n*** without "waiting" for the pipelines to complete a current operation. Likewise, the pipelines ***74_1-74_n*** can retrieve data without "waiting" for the peer to complete a data-sending operation.

[74] Similarly, the output DPSRAM ***102*** includes an input port ***110*** for receiving data from the hardwired pipelines ***74_1-74_n*** via the communication shell ***84***, and includes an output port ***112*** for providing this data to a peer, such as the host processor ***42*** (**FIG. 3**), via the communication interface ***82***. As discussed above, the two data ports ***110*** (input) and ***112*** (output) increase the speed and efficiency of data transfer to/from the DPSRAM ***102***, and using the DPSRAM ***102*** to buffer data from the pipelines ***74_1-74_n*** allows the peer and the pipelines to operate asynchronously relative to one another. That is, the pipelines ***74_1-74_n*** can publish data to the peer without "waiting" for the output-data handler ***126*** to complete a data transfer to the peer or to another peer.

Likewise, the output-data handler ***126*** can transfer data to a peer without "waiting" for the pipelines ***74_1-74_n*** to complete a data-publishing operation.

[75] The working DPSRAM ***104*** includes an input port ***114*** for receiving data from the hardwired pipelines ***74_1-74_n*** via the communication shell ***84***, and includes an output port ***116*** for returning this data back to the pipelines via the communication shell. While processing input data received from the DPSRAM ***100***, the pipelines ***74_1-74_n*** may need to temporarily store partially processed, *i.e.*, intermediate, data before continuing the processing of this data. For example, a first pipeline, such as the pipeline ***74_1***, may generate intermediate data for further processing by a second pipeline, such as the pipeline ***74_2***; thus, the first pipeline may need to temporarily store the intermediate data until the second pipeline retrieves it. The working DPSRAM ***104*** provides this temporary storage. As discussed above, the two data ports ***114*** (input) and ***116*** (output) increase the speed and efficiency of data transfer between the pipelines ***74_1-74_n*** and the DPSRAM ***104***. Furthermore, including a separate working DPSRAM ***104*** typically increases the speed and efficiency of the pipeline circuit ***80*** by allowing the DPSRAMs ***100*** and ***102*** to function exclusively as data-input and data-output buffers, respectively. But, with slight modification to the pipeline circuit ***80***, either or both of the DPSRAMS ***100*** and ***102*** can also be a working memory for the pipelines ***74_1-74_n*** when the DPSRAM ***104*** is omitted, and even when it is present.

[76] Although the DPSRAMS ***100***, ***102***, and ***104*** are described as being external to the pipeline circuit ***80***, one or more of these DPSRAMS, or equivalents thereto, may be internal to the pipeline circuit.

[77] Still referring to **FIG. 5**, the communication interface ***82*** includes an industry-standard bus adapter ***118***, an input-data handler ***120***, input-data and input-event queues ***122*** and ***124***, an output-data handler ***126***, and output-data and output-event queues ***128*** and ***130***. Although the queues ***122***, ***124***, ***128***, and ***130*** are shown as single queues, one or more of these queues may include sub queues (not shown) that allow segregation by, *e.g.,* priority, of the values stored in the queues or of the respective data that these values represent.

[78] The industry-standard bus adapter ***118*** includes the physical layer that allows the transfer of data between the pipeline circuit ***80*** and the pipeline bus ***50*** (**FIG. 4**) via the communication bus ***94***. Therefore, if one wishes to change the parameters of the bus ***94***, then he need only modify the adapter ***118*** and not the entire communication interface ***82***. Where the industry-standard bus interface ***91*** is omitted from the pipeline unit ***78***, , then the adapter ***118*** may be modified to allow the transfer of data directly between the pipeline bus ***50*** and the pipeline circuit ***80***. In this latter implementation, the modified adapter ***118*** includes the functionality of the bus interface ***91***, and one need only modify the adapter ***118*** if he/she wishes to change the parameters of the bus ***50***.

[79] The input-data handler ***120*** receives data from the industry-standard adapter ***118***, loads the data into the DPSRAM ***100*** via the input port ***106***, and generates and stores a pointer to the data and a corresponding data identifier in the input-data queue ***122***. If the data is the payload of a message from a peer, such as the host processor ***42*** (**FIG. 3**), then the input-data handler ***120*** extracts the data from the message before loading the data into the DPSRAM ***100***. The input-data handler ***120*** includes an interface ***132***, which writes the data to the input port ***106*** of the DPSRAM ***100*** and which is further discussed below in conjunction with **FIG. 6**. Alternatively, the input-data handler ***120*** can omit the extraction step and load the entire message into the DPSRAM ***100***.

[80] The input-data handler ***120*** also receives events from the industry-standard bus adapter ***118***, and loads the events into the input-event queue ***124***.

[81] Furthermore, the input-data handler ***120*** includes a validation manager ***134***, which determines whether received data or events are intended for the pipeline circuit ***80***. The validation manager ***134*** may make this determination by analyzing the header (or a portion thereof) of the message that contains the data or the event, by analyzing the type of data or event, or the analyzing the instance identification (*i.e.,* the hardwired pipeline ***74*** for which the data/event is intended) of the data or event.

If the input-data handler ***120*** receives data or an event that is not intended for the pipeline circuit ***80***, then the validation manager ***134*** prohibits the input-data handler from loading the received data/even. Where the peer-vector machine ***40*** includes the router ***61*** (**FIG. 3**) such that the pipeline unit ***78*** should receive only data/events that are intended for the pipeline unit, the validation manager ***134*** may also cause the input-data handler ***120*** to send to the host processor ***42*** (**FIG. 3**) an exception message that identifies the exception (erroneously received data/event) and the peer that caused the exception.

[82] The output-data handler ***126*** retrieves processed data from locations of the DPSRAM ***102*** pointed to by the output-data queue ***128***, and sends the processed data to one or more peers, such as the host processor ***42*** (**FIG. 3**), via the industry-standard bus adapter ***118***. The output-data handler ***126*** includes an interface ***136***, which reads the processed data from the DPSRAM ***102*** via the port ***112***. The interface ***136*** is further discussed below in conjunction with **FIG. 7**.

[83] The output-data handler ***126*** also retrieves from the output-event queue ***130*** events generated by the pipelines $74_1 - 74_n$, and sends the retrieved events to one or more peers, such as the host processor ***42*** (**FIG. 3**) via the industry-standard bus adapter ***118***.

[84] Furthermore, the output-data handler ***126*** includes a subscription manager ***138***, which includes a list of peers, such as the host processor ***42*** (**FIG. 3**), that subscribe to the processed data and to the events; the output-data handler uses this list to send the data/events to the correct peers. If a peer prefers the data/event to be the payload of a message, then the output-data handler ***126*** retrieves the network or bus-port address of the peer from the subscription manager ***138***, generates a header that includes the address, and generates the message from the data/event and the header.

[85] Although the technique for storing and retrieving data stored in the DPSRAMS ***100*** and ***102*** involves the use of pointers and data identifiers, one may modify the input- and output-data handlers ***120*** and ***126*** to implement other data-management techniques. Conventional examples of such data-management techniques include pointers using keys or tokens, input/output control (IOC) block, and spooling.

[86] The communication shell ***84*** includes a physical layer that interfaces the hardwired pipelines ***74**$_1$**-74**$_n$* to the output-data queue ***128***, the controller ***86***, and the DPSRAMs ***100***, ***102***, and ***104***. The shell ***84*** includes interfaces ***140*** and ***142***, and optional interfaces ***144*** and ***146***. The interfaces ***140*** and ***146*** may be similar to the interface ***136***; the interface ***140*** reads input data from the DPSRAM ***100*** via the port ***108***, and the interface ***146*** reads intermediate data from the DPSRAM ***104*** via the port ***116***. The interfaces ***142*** and ***144*** may be similar to the interface ***132***; the interface ***142*** writes processed data to the DPSRAM ***102*** via the port ***110***, and the interface ***144*** writes intermediate data to the DPSRAM ***104*** via the port ***114***.

[87] The controller ***86*** includes a sequence manager ***148*** and a synchronization interface ***150***, which receives one or more synchronization signals SYNC. A peer, such as the host processor ***42*** (**FIG. 3**), or a device (not shown) external to the peer-vector machine ***40*** (**FIG. 3**) may generate the SYNC signal, which triggers the sequence manager ***148*** to activate the hardwired pipelines ***74**$_1$**-74**$_n$* as discussed below and in previously cited U.S. Patent App. Serial No. ___ entitled PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE UNITS AND RELATED COMPUTING MACHINE AND METHOD (Attorney Docket No. 1934-15-3). The synchronization interface ***150*** may also generate a SYNC signal to trigger the pipeline circuit ***80*** or to trigger another peer. In addition, the events from the input-event queue ***124*** also trigger the sequence manager ***148*** to activate the hardwired pipelines ***74**$_1$**-74**$_n$* as discussed below.

[88] The sequence manager ***148*** sequences the hardwired pipelines ***74**$_1$**-74**$_n$* through their respective operations via the communication shell ***84***. Typically, each pipeline ***74*** has at least three operating states: preprocessing, processing, and post processing. During preprocessing, the pipeline ***74***, *e.g.*, initializes its registers and retrieves input data from the DPSRAM ***100***. During processing, the pipeline ***74***, *e.g.*, operates on the retrieved data, temporarily stores intermediate data in the DPSRAM ***104***, retrieves the intermediate data from the DPSRAM ***104***, and operates on the intermediate data to generate result data. During post processing, the pipeline ***74***, *e.g.*, loads the result data into the DPSRAM ***102***. Therefore, the sequence manager ***148*** monitors the operation of the pipelines ***74***$_1$-***74***$_n$ and instructs each pipeline when to begin each of its operating states. And one may distribute the pipeline tasks among the operating states differently than described above. For example, the pipeline ***74*** may retrieve input data from the DPSRAM ***100*** during the processing state instead of during the preprocessing state.

[89] Furthermore, the sequence manager ***148*** maintains a predetermined internal operating synchronization among the hardwired pipelines ***74***$_1$-***74***$_n$. For example, to avoid all of the pipelines ***74***$_1$-***74***$_n$ simultaneously retrieving data from the DPSRAM ***100***, it may be desired to synchronize the pipelines such that while the first pipeline ***74***$_1$ is in a preprocessing state, the second pipeline ***74***$_2$ is in a processing state and the third pipeline ***74***$_3$ is in a post-processing state. Because a state of one pipeline ***74*** may require a different number of clock cycles than a concurrently performed state of another pipeline, the pipelines ***74***$_1$-***74***$_n$ may lose synchronization if allowed to run freely. Consequently, at certain times there may be a "bottle neck," as, for example, multiple pipelines ***74*** simultaneously attempt to retrieve data from the DPSRAM ***100***. To prevent the loss of synchronization and its undesirable consequences, the sequence manager ***148*** allows all of the pipelines ***74*** to complete a current operating state before allowing any of the pipelines to proceed to a next operating state. Therefore, the time that the sequence manager ***148*** allots for a current operating state is long enough to allow the slowest pipeline ***74*** to complete that state. Alternatively, circuitry (not shown) for maintaining a predetermined operating synchronization among the hardwired pipelines ***74***$_1$-***74***$_n$ may be included within the pipelines themselves.

[90] In addition to sequencing and internally synchronizing the hardwired pipelines ***74***$_1$ - ***74***$_n$, the sequence manager ***148*** synchronizes the operation of the pipelines to the operation of other peers, such as the host processor ***42*** (**FIG. 3**), and to the operation of other external devices in response to one or more SYNC signals or to an event in the input-events queue ***124***.

[91] Typically, a SYNC signal triggers a time-critical function but requires significant hardware resources; comparatively, an event typically triggers a non-time-critical function but requires significantly fewer hardware resources. As discussed in previously cited U.S. Patent App. Serial No. ___ entitled PIPELINE ACCELERATOR HAVING MULTIPLE PIPELINE UNITS AND RELATED COMPUTING MACHINE AND METHOD (Attorney Docket No. 1934-15-3), because a SYNC signal is routed directly from peer to peer, it can trigger a function more quickly than an event, which must makes its way through, *e.g.,* the pipeline bus ***50*** (**FIG. 3**), the input-data handler ***120***, and the input-event queue ***124***. But because they are separately routed, the SYNC signals require dedicated circuitry, such as routing lines, buffers, and the SYNC interface ***150***, of the pipeline circuit ***80***. Conversely, because they use the existing data-transfer infrastructure (*e.g.* the pipeline bus ***50*** and the input-data handler ***120***), the events require only the dedicated input-event queue ***124***. Consequently, designers tend to use events to trigger all but the most time-critical functions.

[92] The following is an example of function triggering. Assume that a sonar sensor element (not shown) sends blocks of data to the pipeline unit ***78***, the input-data handler ***120*** stores this data in the DPSRAM ***100***, the pipeline ***74_1*** transfers this data from the DPSRAM ***100*** to the DPSRAM ***104***, and, when triggered, the pipeline ***74_2*** retrieves and processes the data from the DPSRAM ***104***. If the processing that the pipeline ***74_2*** performs on the data is time critical, then the sensor element may generate a SYNC pulse to trigger the pipeline ***74_2***, via the interface ***150*** and the sequence manager ***148***, as soon as the pipeline ***74_1*** finishes loading an entire block of data into the DPSRAM ***104***. There are many conventional techniques that the pipeline unit ***78*** and the sensor can employ to determine when the pipeline ***74_1*** is finished. For example, as discussed below, the sequence manager ***148*** may provide a corresponding SYNC pulse or event to the sensor. Alternatively, if the processing that the pipeline ***74_2*** performs is not time critical, then the sensor may send an event to the sequence manager ***148*** via the pipeline bus ***50*** (**FIG. 3**).

[93] The sequence manager ***148*** may also provide to a peer, such as the host processor ***42*** (**FIG. 3**), information regarding the operation of the hardwired pipelines ***74_1-74_n*** by generating a SYNC pulse or an event. The sequence manager ***148*** sends a SYNC pulse via the SYNC interface ***150*** and a dedicated line (not shown), and sends an event via the output-event queue ***130*** and the output-data handler ***126***. Referring to the above example, suppose that a peer further processes the data blocks from the pipeline ***74_2***. The sequence manager ***148*** may notify the peer via a SYNC pulse or an event when the pipeline ***74_2*** has finished processing a block of data. The sequence manager ***148*** may also confirm receipt of a SYNC pulse or an event by generating and sending a corresponding SYNC pulse or event to the appropriate peer(s).

[94] Still referring to **FIG. 5**, the operation of the pipeline unit ***78*** is discussed according to an embodiment of the invention.

[95] For data, the industry-standard bus interface ***91*** receives data signals (which originates from a peer, such as the host processor ***42*** of **FIG. 3**) from the pipeline bus ***50*** (and the router ***61*** if present), and translates these signals into messages each having a header and payload.

[96] Next, the industry-standard bus adapter ***118*** converts the messages from the industry-standard bus interface ***91*** into a format that is compatible with the input-data handler ***120***.

[97] Then, the input-data handler ***120*** dissects the message headers and extracts from each header the portion that describes the data payload. For example, the extracted header portion may include, *e.g.*, the address of the pipeline unit ***78***, the type of data in the payload, or an instance identifier that identifies the pipeline(s) ***78_1 – 78_n*** for which the data is intended.

[98] Next, the validation manager ***134*** analyzes the extracted header portion and confirms that the data is intended for one of the hardwired pipelines ***74_1-74_n***, the interface ***132*** writes the data to a location of the DPSRAM ***100*** via the port ***106***, and the input-data handler ***120*** stores a pointer to the location and a corresponding data identifier in the input-data queue ***122***. The data identifier identifies the pipeline or

pipelines ***74$_1$*-*74$_n$*** for which the data is intended, or includes information that allows the sequence manager ***148*** to make this identification as discussed below. Alternatively, the queue ***122*** may include a respective subqueue (not shown) for each pipeline ***74$_1$*-*74$_n$***, and the input-data handler ***120*** stores the pointer in the subqueue or subqueues of the intended pipeline or pipelines. In this alternative, the data identifier may be omitted. Furthermore, if the data is the payload of a message, then the input-data handler ***120*** extracts the data from the message before the interface ***132*** stores the data in the DPSRAM ***100***. Alternatively, as discussed above, the interface ***132*** may store the entire message in the DPSRAM ***100***.

[99] Then, at the appropriate time, the sequence manager ***148*** reads the pointer and the data identifier from the input-data queue ***122***, determines from the data identifier the pipeline or pipelines ***74$_1$* – *74$_n$*** for which the data is intended, and passes the pointer to the pipeline or pipelines via the communication shell ***84***.

[100] Next, the data-receiving pipeline or pipelines ***74$_1$* – *74$_n$*** cause the interface ***140*** to retrieve the data from the pointed-to location of the DPSRAM ***100*** via the port ***108***.

[101] Then, the data-receiving pipeline or pipelines ***74$_1$*-*74$_n$*** process the retrieved data, the interface ***142*** writes the processed data to a location of the DPSRAM ***102*** via the port ***110***, and the communication shell ***84*** loads into the output-data queue ***128*** a pointer to and a data identifier for the processed data. The data identifier identifies the destination peer or peers, such as the host processor ***42*** (**FIG. 3**), that subscribe to the processed data, or includes information (such as the data type) that allows the subscription manager ***138*** to subsequently determine the destination peer or peers (*e.g.*, the host processor ***42*** of **FIG. 3**). Alternatively, the queue ***128*** may include a respective subqueue (not shown) for each pipeline ***74$_1$*-*74$_n$***, and the communication shell ***84*** stores the pointer in the subqueue or subqueues of the originating pipeline or pipelines. In this alternative, the communication shell ***84*** may omit loading a data identifier into the queue ***128***. Furthermore, if the pipeline or pipelines ***74$_1$*-*74$_n$*** generate intermediate data while processing the retrieved data, then the interface ***144*** writes the intermediate data into the DPSRAM ***104*** via the port ***114***, and the interface ***146*** retrieves the intermediate data from the DPSRAM ***104*** via the port ***116***.

[102] Next, the output-data handler ***126*** retrieves the pointer and the data identifier from the output-data queue ***128***, the subscription manager ***138*** determines from the identifier the destination peer or peers (*e.g.*, the host processor ***42*** of **FIG. 3**) of the data, the interface ***136*** retrieves the data from the pointed-to location of the DPSRAM ***102*** via the port ***112***, and the output-data handler sends the data to the industry-standard bus adapter ***118***. If a destination peer requires the data to be the payload of a message, then the output-data handler ***126*** generates the message and sends the message to the adapter ***118***. For example, suppose the data has multiple destination peers and the pipeline bus ***50*** supports message broadcasting. The output-data handler ***126*** generates a single header that includes the addresses of all the destination peers, combines the header and data into a message, and sends (via the adapter ***118*** and the industry-standard bus interface ***91***) a single message to all of the destination peers simultaneously. Alternatively, the output-data handler ***126*** generates a respective header, and thus a respective message, for each destination peer, and sends each of the messages separately.

[103] Then, the industry-standard bus adapter ***118*** formats the data from the output-data handler ***126*** so that it is compatible with the industry-standard bus interface ***91***.

[104] Next, the industry-standard bus interface ***91*** formats the data from the industry-standard bus adapter ***118*** so that it is compatible with the pipeline bus ***50*** (**FIG. 3**).

[105] For an event with no accompanying data, *i.e.,* a doorbell, the industry-standard bus interface ***91*** receives a signal (which originates from a peer, such as the host processor ***42*** of **FIG. 3**) from the pipeline bus ***50*** (and the router ***61*** if present), and translates the signal into a header (*i.e.*, a data-less message) that includes the event.

[106] Next, the industry-standard bus adapter ***118*** converts the header from the industry-standard bus interface ***91*** into a format that is compatible with the input-data handler ***120***.

[107] Then, the input-data handler ***120*** extracts from the header the event and a description of the event. For example, the description may include, *e.g.*, the address of the pipeline unit ***78***, the type of event, or an instance identifier that identifies the pipeline(s) ***78_1 – 78_n*** for which the event is intended.

[108] Next, the validation manager ***134*** analyzes the event description and confirms that the event is intended for one of the hardwired pipelines ***74_1-74_n***, and the input-data handler ***120*** stores the event and its description in the input-event queue ***124***.

[109] Then, at the appropriate time, the sequence manager ***148*** reads the event and its description from the input-event queue ***124***, and, in response to the event, triggers the operation of one or more of the pipelines ***74_1-74_n*** as discussed above. For example, the sequence manager ***148*** may trigger the pipeline ***74_2*** to begin processing data that the pipeline ***74_1*** previously stored in the DPSRAM ***104***.

[110] To output an event, the sequence manager ***148*** generates the event and a description of the event, and loads the event and its description into the output-event queue ***130*** — the event description identifies the destination peer(s) for the event if there is more than one possible destination peer. For example, as discussed above, the event may confirm the receipt and implementation of an input event, an input-data or input-event message, or a SYNC pulse

[111] Next, the output-data handler ***126*** retrieves the event and its description from the output-event queue ***130***, the subscription manager ***138*** determines from the event description the destination peer or peers (*e.g.*, the host processor ***42*** of **FIG. 3**) of the event, and the output-data handler sends the event to the proper destination peer or peers via the industry-standard bus adapter ***118*** and the industry-standard bus interface ***91*** as discussed above.

[112] For a configuration command, the industry-standard bus adapter ***118*** receives the command from the host processor ***42*** (**FIG. 3**) via the industry-standard

bus interface ***91***, and provides the command to the input-data handler ***120*** in a manner similar to that discussed above for a data-less event (*i.e.*, doorbell)

[113] Next, the validation manager ***134*** confirms that the command is intended for the pipeline unit ***78***, and the input-data handler ***120*** loads the command into the configuration manager ***90***. Furthermore, either the input-data handler ***120*** or the configuration manager ***90*** may also pass the command to the output-data handler ***126***, which confirms that the pipeline unit ***78*** received the command by sending the command back to the peer (*e.g.*, the host processor ***42*** of **FIG. 3**) that sent the command. This confirmation technique is sometimes called "echoing."

[114] Then, the configuration manager ***90*** implements the command. For example, the command may cause the configuration manager ***90*** to disable one of the pipelines $\boldsymbol{74_1}$-$\boldsymbol{74_n}$ for debugging purposes. Or, the command may allow a peer, such as the host processor ***42*** (**FIG. 3**), to read the current configuration of the pipeline circuit ***80*** from the configuration manager ***90*** via the output-data handler ***126***. In addition, one may use a configuration command to define an exception that is recognized by the exception manager ***88***.

[115] For an exception, a component, such as the input-data queue ***122***, of the pipeline circuit ***80*** triggers an exception to the exception manager ***88***. In one implementation, the component includes an exception-triggering adapter (not shown) that monitors the component and triggers the exception in response to a predetermined condition or set of conditions. The exception-triggering adapter may be a universal circuit that can be designed once and then included as part of each component of the pipeline circuit ***80*** that generates exceptions.

[116] Next, in response to the exception trigger, the exception manager ***88*** generates an exception identifier. For example, the identifier may indicate that the input-data queue ***122*** has overflowed. Furthermore, the identifier may include its destination peer if there is more than one possible destination peer.

[117] Then, the output-data handler ***126*** retrieves the exception identifier from the exception manager ***88*** and sends the exception identifier to the host processor ***42***

(**FIG. 3**) as discussed in previously cited U.S. Patent App. Serial No. _____ entitled COMPUTING MACHINE HAVING IMPROVED COMPUTING ARCHITECTURE AND RELATED SYSTEM AND METHOD (Attorney Docket No. 1934-12-3). Alternatively, if there are multiple possible destination peers, then the exception identifier can also include destination information from which the subscription manager ***138*** determines the destination peer or peers (*e.g.*, the host processor ***42*** of **FIG. 3**) of the identifier. The output-data handler ***126*** then sends the identifier to the destination peer or peers via the industry-standard bus adapter ***118*** and the industry-standard bus interface ***91***.

[118] Still referring to **FIG. 5**, alternative embodiments to the pipeline unit ***78*** exist. For example, although described as including DPSRAMs, the data memory ***92*** may include other types of memory ICs such as quad-data-rate (QDR) SRAMs.

[119] **FIG. 6** is a block diagram of the interface ***142*** of **FIG. 5** according to an embodiment of the invention. As discussed above in conjunction with **FIG. 5**, the interface ***142*** writes processed data from the hardwired pipelines 74_1-74_n to the DPSRAM ***102***. As discussed below, the structure of the interface ***142*** reduces or eliminates data "bottlenecks" and, where the pipeline circuit ***80*** (**FIG. 5**) is a PLIC, makes efficient use of the PLIC's local and global routing resources.

[120] The interface ***142*** includes write channels $150_1 – 150_n$, one channel for each hardwired pipeline $74_1 – 74_n$ (**FIG. 5**), and includes a controller ***152***. For purposes of illustration, the channel 150_1 is discussed below, it being understood that the operation and structure of the other channels $150_2 – 150_n$ are similar unless stated otherwise.

[121] The channel 150_1 includes a write-address/data FIFO 154_1 and a address/data register 156_1.

[122] The FIFO 154_1 stores the data that the pipeline 74_1 writes to the DPSRAM ***102***, and stores the address of the location within the DPSRAM ***102*** to which the pipeline writes the data, until the controller ***152*** can actually write the data to the DPSRAM ***102*** via the register 156_1. Therefore, the FIFO 154_1 reduces or eliminates the data bottleneck that may occur if the pipeline ***74_1*** had to “wait” to write data to the channel ***150_1*** until the controller ***152*** finished writing previous data.

[123] The FIFO ***154_1*** receives the data from the pipeline ***74_1*** via a bus ***158_1***, receives the address of the location to which the data is to be written via a bus ***160_1***, and provides the data and address to the register ***156_1*** via busses ***162_1*** and ***164_1***, respectively. Furthermore, the FIFO ***154_1*** receives a WRITE FIFO signal from the pipeline ***74_1*** on a line ***166_1***, receives a CLOCK signal via a line ***168_1***, and provides a FIFO FULL signal to the pipeline ***74_1*** on a line ***170_1***. In addition, the FIFO ***154_1*** receives a READ FIFO signal from the controller ***152*** via a line ***172_1***, and provides a FIFO EMPTY signal to the controller via a line ***174_1***. Where the pipeline circuit ***80*** (**FIG. 5**) is a PLIC, the busses ***158_1***, ***160_1***, ***162_1***, and ***164_1*** and the lines ***166_1***, ***168_1***, ***170_1***, ***172_1***, and ***174_1*** are preferably formed using local routing resources. Typically, local routing resources are preferred to global routing resources because the signal-path lengths are generally shorter and the routing is easier to implement.

[124] The register ***156_1*** receives the data to be written and the address of the write location from the FIFO ***154_1*** via the busses ***162_1*** and ***164_1***, respectively, and provides the data and address to the port ***110*** of the DPSRAM ***102*** (**FIG. 5**) via an address/data bus ***176***. Furthermore, the register ***156_1*** also receives the data and address from the registers ***156_2 – 156_n*** via an address/data bus ***178_1*** as discussed below. In addition, the register ***156_1*** receives a SHIFT/LOAD signal from the controller ***152*** via a line ***180***. Where the pipeline circuit ***80*** (**FIG. 5**) is a PLIC, the bus ***176*** is typically formed using global routing resources, and the busses ***178_1 – 178_{n-1}*** and the line ***180*** are preferably formed using local routing resources.

[125] In addition to receiving the FIFO EMPTY signal and generating the READ FIFO and SHIFT/LOAD signals, the controller ***152*** provides a WRITE DPSRAM signal to the port ***110*** of the DPSRAM ***102*** (**FIG. 5**) via a line ***182***.

[126] Still referring to **FIG. 6**, the operation of the interface ***142*** is discussed.

[127] First, the FIFO ***154_1*** drives the FIFO FULL signal to the logic level corresponding to the current state (“full” or “not full”) of the FIFO.

[128] Next, if the FIFO ***154₁*** is not full and the pipeline ***74₁*** has processed data to write, the pipeline drives the data and corresponding address onto the busses ***158₁*** and ***160₁***, respectively, and asserts the WRITE signal, thus loading the data and address into the FIFO. If the FIFO ***154₁*** is full, however, the pipeline ***74₁*** waits until the FIFO is not full before loading the data.

[129] Then, the FIFO ***154₁*** drives the FIFO EMPTY signal to the logic level corresponding to the current state ("empty" or "not empty") of the FIFO.

[130] Next, if the FIFO ***154₁*** is not empty, the controller ***152*** asserts the READ FIFO signal and drives the SHIFT/LOAD signal to the load logic level, thus loading the first loaded data and address from the FIFO into the register ***156₁***. If the FIFO ***154₁*** is empty, the controller ***152*** does not assert READ FIFO, but does drive SHIFT load to the load logic level if any of the other FIFOs ***154₂-154ₙ*** are not empty.

[131] The channels ***150₂ – 150ₙ*** operate in a similar manner such that first-loaded data in the FIFOs ***154₂ – 154ₙ*** are respectively loaded into the registers ***156₂-156ₙ***.

[132] Then, the controller ***152*** drives the SHIFT/LOAD signal to the shift logic level and asserts the WRITE DPSRAM signal, thus serially shifting the data and addresses from the registers ***156₁ – 156ₙ*** onto the address/data bus ***176*** and loading the data into the corresponding locations of the DPSRAM ***102***. Specifically, during a first shift cycle, the data and address from the register ***156₁*** are shifted onto the bus ***176*** such that the data from the FIFO ***154₁*** is loaded into the addressed location of the DPSRAM ***102***. Also during the first shift cycle, the data and address from the register ***156₂*** are shifted into the register ***156₁***, the data and address from the register ***156₃*** (not shown) are shifted into the register ***156₂***, and so on. During a second shift cycle, the data and address from the register ***156₁*** are shifted onto the bus ***176*** such that the data from the FIFO ***154₂*** is loaded into the addressed location of the DPSRAM ***102***. Also during the second shift cycle, the data and address from the register ***156₂*** are shifted into the register ***156₁***, the data and address from the register ***156₃*** (not shown) are shifted into the register ***156₂***, and so on. There are n shift cycles, and during the nth shift cycle the data and address from the register ***156_n*** (which is the data and address from the FIFO ***154_n***) is shifted onto the bus ***176***. The controller ***152*** may implement these shift cycles by pulsing the SHIFT/LOAD signal, or by generating a shift clock signal (not shown) that is coupled to the registers ***156_1-156_n***. Furthermore, if one of the registers ***156_1-156_n*** is empty during a particular shift operation because its corresponding FIFO ***154_1-154_n*** was empty when the controller ***152*** loaded the register, then the controller may bypass the empty register, and thus shorten the shift operation by avoiding shifting null data and a null address onto the bus ***176***.

[133] Referring to **FIGS. 5** and **6**, according to an embodiment of the invention, the interface ***144*** is similar to the interface ***142***, and the interface ***132*** is also similar to the interface ***142*** except that the interface ***132*** includes only one write channel ***150***.

[134] **FIG. 7** is a block diagram of the interface ***140*** of **FIG. 5** according to an embodiment of the invention. As discussed above in conjunction with **FIG. 5**, the interface ***140*** reads input data from the DPSRAM ***100*** and transfers this data to the hardwired ***74_1-74_n***. As discussed below, the structure of the interface ***140*** reduces or eliminates data "bottlenecks" and, where the pipeline circuit ***80*** (**FIG. 5**) is a PLIC, makes efficient use of the PLIC's local and global routing resources.

[135] The interface ***140*** includes read channels ***190_1 – 190_n***, one channel for each hardwired pipeline ***74_1 – 74_n*** (**FIG. 5**), and a controller ***192***. For purposes of illustration, the read channel ***190_1*** is discussed below, it being understood that the operation and structure of the other read channels ***190_2 – 190_n*** are similar unless stated otherwise.

[136] The channel ***190_1*** includes a FIFO ***194_1*** and an address/identifier (ID) register ***196_1***. As discussed below, the identifier identifies the pipeline ***74_1-74_n*** that makes the request to read data from a particular location of the DPSRAM ***100*** to receive the data.

[137] The FIFO ***194_1*** includes two sub-FIFOs (not shown), one for storing the address of the location within the DPSRAM ***100*** from which the pipeline ***74_1*** wishes to read the input data, and the other for storing the data read from the DPSRAM ***100***.

Therefore, the FIFO ***194₁*** reduces or eliminates the bottleneck that may occur if the pipeline ***74₁*** had to "wait" to provide the read address to the channel ***190₁*** until the controller ***192*** finished reading previous data, or if the controller had to wait until the pipeline ***74₁*** retrieved the read data before the controller could read subsequent data.

[138] The FIFO ***194₁*** receives the read address from the pipeline ***74₁*** via a bus ***198₁*** and provides the address and ID to the register ***196₁*** via a bus ***200₁***. Since the ID corresponds to the pipeline ***74₁*** and typically does not change, the FIFO ***194₁*** may store the ID and concatenate the ID with the address. Alternatively, the pipeline ***74₁*** may provide the ID to the FIFO ***194₁*** via the bus ***198₁***. Furthermore, the FIFO ***194₁*** receives a READY WRITE FIFO signal from the pipeline ***74₁*** via a line ***202₁***, receives a CLOCK signal via a line ***204₁***, and provides a FIFO FULL (of read addresses) signal to the pipeline via a line ***206₁***. In addition, the FIFO ***194₁*** receives a WRITE/READ FIFO signal from the controller ***192*** via a line ***208₁***, and provides a FIFO EMPTY signal to the controller via a line ***210₁***. Moreover, the FIFO ***194₁*** receives the read data and the corresponding ID from the controller ***192*** via a bus ***212***, and provides this data to the pipeline ***74₁*** via a bus ***214₁***. Where the pipeline circuit ***80*** (**FIG. 5**) is a PLIC, the busses ***198₁***, ***200₁***, and ***214₁*** and the lines ***202₁***, ***204₁***, ***206₁***, ***208₁***, and ***210₁*** are preferably formed using local routing resources, and the bus ***212*** is typically formed using global routing resources.

[139] The register ***196₁*** receives the address of the location to be read and the corresponding ID from the FIFO ***194₁*** via the bus ***206₁***, provides the address to the port ***108*** of the DPSRAM ***100*** (**FIG. 5**) via an address bus ***216***, and provides the ID to the controller ***192*** via a bus ***218***. Furthermore, the register ***196₁*** also receives the addresses and IDs from the registers ***196₂ – 196ₙ*** via an address/ID bus ***220₁*** as discussed below. In addition, the register ***196₁*** receives a SHIFT/LOAD signal from the controller ***192*** via a line ***222***. Where the pipeline circuit ***80*** (**FIG. 5**) is a PLIC, the bus ***216*** is typically formed using global routing resources, and the busses ***220₁-220ₙ₋₁*** and the line ***222*** are preferably formed using local routing resources.

[140] In addition to receiving the FIFO EMPTY signal, generating the WRITE/READ FIFO and SHIFT/LOAD signals, and providing the read data and corresponding ID, the controller ***192*** receives the data read from the port ***108*** of the DPSRAM ***100*** (**FIG. 5**) via a bus ***224*** and generates a READ DPSRAM signal on a line ***226***, which couples this signal to the port ***108***. Where the pipeline circuit ***80*** (**FIG. 5**) is a PLIC, the bus ***224*** and the line ***226*** are typically formed using global routing resources.

[141] Still referring to **FIG. 7**, the operation of the interface ***140*** is discussed.

[142] First, the FIFO ***194***$_1$ drives the FIFO FULL signal to the logic level corresponding to the current state ("full" or "not full") of the FIFO relative to the read addresses. That is, if the FIFO ***194***$_1$ is full of addresses to be read, then it drives the logic level of FIFO FULL to one level, and if the FIFO is not full of read addresses, it drives the logic level of FIFO FULL to another level.

[143] Next, if the FIFO ***194***$_1$ is not full of read addresses and the pipeline ***74***$_1$ is ready for more input data to process, the pipeline drives the address of the data to be read onto the bus ***198***$_1$, and asserts the READ/WRITE FIFO signal to a write level, thus loading the address into the FIFO. As discussed above in conjunction with **FIG. 5**, the pipeline ***74***$_1$ gets the address from the input-data queue ***122*** via the sequence manager ***148***. If, however, the FIFO ***194***$_1$ is full of read addresses, the pipeline ***74***$_1$ waits until the FIFO is not full before loading the read address.

[144] Then, the FIFO ***194***$_1$ drives the FIFO EMPTY signal to the logic level corresponding to the current state ("empty" or "not empty") of the FIFO relative to the read addresses. That is, if the FIFO ***194***$_1$ is loaded with at least one read address, it drives the logic level of FIFO EMPTY to one level, and if the FIFO is loaded with no read addresses, it drives the logic level of FIFO EMPTY to another level.

[145] Next, if the FIFO ***194***$_1$ is not empty, the controller ***192*** asserts the WRITE/READ FIFO signal to the read logic level and drives the SHIFT/LOAD signal to the load logic level, thus loading the first loaded address and the ID from the FIFO into the register ***196***$_1$.

[146] The channels ***190***$_2$ – ***190***$_n$ operate in a similar manner such that the controller ***192*** respectively loads the first-loaded addresses and IDs from the FIFOs ***194***$_2$ – ***194***$_n$ into the registers ***196***$_2$ - ***196***$_n$. If all of the FIFOs ***194***$_2$-***194***$_n$ are empty, then the controller ***192*** waits for at least one of the FIFOs to receive an address before proceeding.

[147] Then, the controller ***192*** drives the SHIFT/LOAD signal to the shift logic level and asserts the READ DPSRAM signal to serially shift the addresses and IDs from the registers ***196***$_1$ – ***196***$_n$ onto the address and ID busses ***216*** and ***218*** and to serially read the data from the corresponding locations of the DPSRAM ***100*** via the bus ***224***.

[148] Next, the controller ***192*** drives the received data and corresponding ID — the ID allows each of the FIFOs ***194***$_1$ – ***194***$_n$ to determine whether it is an intended recipient of the data — onto the bus ***212***, and drives the WRITE/READ FIFO signal to a write level, thus serially writing the data to the respective FIFO, ***194***$_1$-***194***$_n$.

[149] Then, the hardwired pipelines ***74***$_1$-***74***$_n$ sequentially assert their READ/WRITE FIFO signals to a read level and sequentially read the data via the busses ***214***$_1$-***214***$_n$.

[150] Still referring to **FIG. 7**, a more detailed discussion of their data-read operator is presented.

[151] During a first shift cycle, the controller ***192*** shifts the address and ID from the register ***196***$_1$ onto the busses ***216*** and ***218***, respectively, asserts read DPSRAM, and thus reads the data from the corresponding location of the DPSRAM ***100*** via the bus ***224*** and reads the ID from the bus ***218***. Next, the controller ***192*** drives WRITE/READ FIFO signal on the line ***208***$_1$ to a write level and drives the received data and the ID onto the bus ***212***. Because the ID is the ID from the FIFO ***194***$_1$, the FIFO ***194***$_1$ recognizes the ID and thus loads the data from the bus ***212*** in response the write level of the WRITE/READ FIFO signal. The remaining FIFOs ***194***$_2$ – ***194***$_n$ do not load the data because the ID on the bus ***212*** does not correspond to their IDs. Then, the pipeline ***74***$_1$ asserts the READ/WRITE FIFO signal on the line ***202***$_1$ to the read level and retrieves the read data via the bus ***214***$_1$. Also during the first shift cycle, the address and ID from the register ***196_2*** are shifted into the register ***196_1***, the address and ID from the register ***196_3*** (not shown) are shifted into the register ***196_2***, and so on. Alternatively, the controller ***192*** may recognize the ID and drive only the WRITE/READ FIFO signal on the line ***208_1*** to the write level. This eliminates the need for the controller ***192*** to send the ID to the FIFOs ***194_1-194_n***. In another alternative, the WRITE/READ FIFO signal may be only a read signal, and the FIFO ***194_1*** (as well as the other FIFOs ***194_2-194_n***) may load the data on the bus ***212*** when the ID on the bus ***212*** matches the ID of the FIFO ***194_1***. This eliminates the need of the controller ***192*** to generate a write signal.

[152] During a second shift cycle, the address and ID from the register ***196_1*** is shifted onto the busses ***216*** and ***218*** such that the controller ***192*** reads data from the location of the DPSRAM ***100*** specified by the FIFO ***194_2***. Next, the controller ***192*** drives the WRITE/READ FIFO signal to a write level and drives the received data and the ID onto the bus ***212***. Because the ID is the ID from the FIFO ***194_2***, the FIFO ***194_2*** recognizes the ID and thus loads the data from the bus ***212***. The remaining FIFOs ***194_1*** and ***194_3 – 194_n*** do not load the data because the ID on the bus ***212*** does not correspond to their IDs. Then, the pipeline ***74_2*** asserts its READ/WRITE FIFO signal to the read level and retrieves the read data via the bus ***214_2***. Also during the second shift cycle, the address and ID from the register ***196_2*** is shifted into the register ***196_1***, the address and ID from the register ***196_3*** (not shown) is shifted into the register ***196_2***, and so on.

[153] This continues for n shift cycles, *i.e.,* until the address and ID from the register ***196_n*** (which is the address and ID from the FIFO ***194_n***) are respectively shifted onto the bus ***216*** and ***218***. The controller ***192*** may implement these shift cycles by pulsing the SHIFT/LOAD signal, or by generating a shift clock signal (not shown) that is coupled to the registers ***196_1-196_n***. Furthermore, if one of the registers ***196_1-196_2*** is empty during a particular shift operation because its corresponding FIFO ***194_1-194_n*** is empty, then the controller ***192*** may bypass the empty register, and thus shorten the shift operation by avoiding shifting a null address onto the bus ***216***.

[154] Referring to **FIGS. 5** and **6**, according to an embodiment of the invention, the interface ***144*** is similar to the interface ***140***, and the interface ***136*** is also similar to the interface ***140*** except that the interface ***136*** includes only one read channel ***190***, and thus includes no ID circuitry.

[155] **Fig. 8** is a schematic block diagram of a pipeline unit ***230*** of **FIG. 4** according to another embodiment of the invention. The pipeline unit ***230*** is similar to the pipeline unit ***78*** of **FIG. 4** except that the pipeline unit ***230*** includes multiple pipeline circuits ***80*** — here two pipeline circuits ***80a*** and ***80b***. Increasing the number of pipeline circuits ***80*** typically allows an increase in the number n of hardwired pipelines ***74_1-74_n***, and thus an increase in the functionality of the pipeline unit ***230*** as compared to the pipeline unit ***78***.

[156] In the pipeline unit ***230*** of **FIG. 8**, the services components, *i.e.*, the communication interface ***82***, the controller ***86***, the exception manager ***88***, the configuration manager ***90***, and the optional industry-standard bus interface ***91***, are disposed on the pipeline circuit ***80a***, and the pipelines ***74_1-74_n*** and the communication shell ***84*** are disposed on the pipeline circuit ***80b***. By locating the services components and the pipelines ***74_1-74_n*** on separate pipeline circuits, one can include a higher number n of pipelines and/or more complex pipelines than he can where the service components and the pipelines are located on the same pipeline circuit. Alternatively, the portion of the communication shell ***84*** that interfaces the pipelines ***74_1-74_n*** to the interface ***82*** and the controller ***86*** may be disposed on the pipeline circuit ***80a***.

[157] **FIG. 9** is a schematic block diagram of the pipeline circuits ***80a*** and ***80b*** and the data memory ***92*** of the pipeline unit ***230*** of **FIG. 8** according to an embodiment of the invention. Other than the pipeline components being disposed on two pipelines circuits, the structure and operation of the pipeline circuits ***80a*** and ***80b*** and the memory ***92*** of **FIG. 9** are the same as for the pipeline circuit ***80*** and memory ***92*** of **FIG. 5**.

[158] The preceding discussion is presented to enable a person skilled in the art to make and use the invention. Various modifications to the embodiments will be readily apparent to those skilled in the art, and the generic principles herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.